NBSI v1.0 — Engineering Build Plan

Lightweight Stack

Full Build Plan for CPU-Only Hardware

spaCy extraction · MiniLM embedding · NBSI core (unchanged) · llama-cpp synthesis · OS file watcher

Target hardware	Raspberry Pi 5 — 8 GB RAM · ARM Cortex-A76 · no GPU required
Extends	nbsi_v1_external.zip — all 33 tests still pass unchanged
New components	spaCy extractor · RealEmbedder · Synthesiser · OS watcher
Disk footprint	~2.5 GB total (models + deps + code)
Ingestion speed	~2–4 document pages per minute on Pi 5 (CPU)
Query latency	Graph traversal < 100 ms · Synthesis 20–60 s depending on model
Engineer-days	3–5 days to working beta

0. What This Plan Builds

This document is the engineer's build plan for the NBSI lightweight stack — the complete system that runs on a Raspberry Pi 5 with no GPU. It connects four new components to the existing NBSI v1.0 core. Nothing in the core changes. The core is already built and already tested. This plan wires real inputs and real outputs to the architecture that is already running.

Layer	Component	Status	This plan
Extraction	spaCy NLP → nodes + edges	Not built	Section 2
Embedding	MiniLM-L6-v2 sentence transformer	Stub only	Section 3
Graph + lifecycle	Full NBSI core (all modules)	✓ Complete, 33/33 tests	Unchanged
Synthesis	llama-cpp 1–3 B quantised model	Not built	Section 4
OS integration	File watcher + document reader	Not built	Section 5
Orchestration	Main entry point + persistence	Not built	Section 6

PREREQUISITES BEFORE STARTING

Download nbsi_v1_external.zip. Install networkx, numpy, scipy. Run python3 run_tests.py and confirm 33 tests pass. Every section in this plan assumes that foundation is working. Do not skip this verification step — it is the only way to confirm the environment is correct before adding dependencies.

1. Environment Setup

1.1 Hardware

Component	Minimum	Recommended	Notes
RAM	4 GB	8 GB	1.5 B model needs ~2.5 GB total; 3 B needs ~5 GB
CPU	Pi 4 — Cortex-A55	Pi 5 — Cortex-A76	Pi 5 is ~2.5× faster for llama-cpp inference
Storage	16 GB SD or SSD	32 GB SSD	SSD strongly preferred — models are write-heavy on first load
OS	Raspberry Pi OS 64-bit	Bookworm 64-bit	Must be 64-bit for llama-cpp to compile correctly
Python	3.10	3.11	3.12 has llama-cpp build issues as of early 2026

1.2 System packages

```
sudo apt update && sudo apt upgrade -y

# Build tools required for llama-cpp compilation
sudo apt install -y build-essential cmake git

# Python 3.11
sudo apt install -y python3.11 python3.11-venv python3.11-dev

# Create and activate virtual environment
python3.11 -m venv ~/nbsi-env
source ~/nbsi-env/bin/activate

# Confirm version
python --version   # Must show 3.11.x
```

1.3 Python packages

```
# NBSI core deps (already in requirements.txt)
pip install networkx numpy scipy

# Extraction
pip install spacy
python -m spacy download en_core_web_sm      # 12 MB — fast, sufficient for most prose
# python -m spacy download en_core_web_trf   # 500 MB — transformer-based, more
accurate

# Embedding
pip install sentence-transformers            # MiniLM downloads on first use (~80 MB)
```

```
# Synthesis — compiles llama.cpp from source, takes 10–15 min on Pi 5
pip install llama-cpp-python

# OS integration
pip install watchdog                        # File system events
pip install pdfminer.six                    # PDF reading
pip install python-docx                     # DOCX reading

# Run the existing tests to confirm nothing broke
cd ~/nbsi && PYTHONPATH=. python3 run_tests.py   # Must still show 33/33
```

1.4 Model downloads

Models are downloaded once and stored permanently. They do not update. Choose one synthesis model based on available RAM.

```
mkdir -p ~/nbsi-models

# Pre-download MiniLM (80 MB) to avoid runtime delay
python -c "from sentence_transformers import SentenceTransformer; \
           SentenceTransformer('sentence-transformers/all-MiniLM-L6-v2')"

# SYNTHESIS MODEL — choose one:

# Option A: Qwen2.5 1.5B Q4 (~900 MB) — best balance, fits in 4 GB RAM
wget -P ~/nbsi-models \
  https://huggingface.co/Qwen/Qwen2.5-1.5B-Instruct-GGUF/resolve/main/qwen2.5-1.5b-
instruct-q4_k_m.gguf

# Option B: Phi-3 Mini 3.8B Q4 (~2.2 GB) — better quality, needs 6 GB RAM
wget -P ~/nbsi-models \
  https://huggingface.co/microsoft/Phi-3-mini-4k-instruct-gguf/resolve/main/Phi-3-
mini-4k-instruct-q4.gguf

# Option C: Qwen2.5 0.5B Q4 (~400 MB) — minimum viable, runs in 2 GB RAM
wget -P ~/nbsi-models \
  https://huggingface.co/Qwen/Qwen2.5-0.5B-Instruct-GGUF/resolve/main/qwen2.5-0.5b-
instruct-q4_k_m.gguf
```

MODEL SELECTION

Start with Qwen2.5 1.5B Q4. It fits in 4 GB RAM alongside all other NBSI components, produces coherent path narration, and runs at roughly 10–15 tokens per second on a Pi 5. Upgrade to Phi-3 Mini only if synthesis quality is genuinely inadequate for your use case — it costs roughly 3× the RAM and 3× the latency.

2. Extraction Layer — spaCy Pipeline

The extraction pipeline reads plain text and produces ConceptNodes and ConceptEdges that are fed directly into NBSISession.ingest_graph(). This replaces the LLM-based extraction in the full NBSI spec with a CPU-only NLP approach. No API calls, no GPU, no cost per document.

2.1 What spaCy provides

spaCy feature	What it finds	Maps to NBSI
Named Entity Recognition	People, places, organisations, dates, quantities	ConceptNode node_type='entity'
Noun chunks	Compound concepts: 'global warming', 'carbon tax'	ConceptNode node_type='concept'
Dependency parse	Subject–verb–object structure within sentences	ConceptEdge between source and target
Sentence boundaries	Clause-level segmentation	Sequential edge grouping
POS tags + lemmas	Verb type helps infer causal vs descriptive edges	Edge type classification

2.2 Edge type inference rules

spaCy does not label relationships as causal or contrastive. Edge type is inferred from transition words and verb lemmas found within the sentence containing the two concepts.

Signal found in sentence	Example	Edge type assigned
Causal verb: cause, lead, produce, drive, trigger, generate	'emissions cause warming'	causal
Marker: because, since, therefore, thus, hence, consequently	'prices rise because demand increases'	causal
Marker: however, but, although, despite, whereas, yet	'warming continues despite reductions'	contrastive
Marker: for example, such as, including, namely, consists of	'renewables such as solar'	constitutive
Marker: before, after, then, subsequently, first, next	'extraction then refinement occurs'	sequential
Definition pattern: X is Y / X means Y / X refers to Y	'photosynthesis is the process'	constitutive
No pattern matched	any other sentence structure	semantic

2.3 Implementation

```
# nbsi/ingestion/spacy_extractor.py
import spacy
from nbsi.graph.node import ConceptNode
from nbsi.graph.edge import ConceptEdge

CAUSAL_VERBS    =
{'cause','lead','produce','drive','trigger','generate','result','create'}
CAUSAL_MARKS    = {'because','since','therefore','thus','hence','consequently'}
CONTRAST_MARKS  =
{'however','but','although','despite','yet','whereas','nevertheless'}
CONSTIT_MARKS   = {'such as','including','for example','namely','consists of'}
SEQUENCE_MARKS  = {'before','after','then','subsequently','first','next','finally'}

class SpacyExtractor:
    def __init__(self, model='en_core_web_sm'):
        self.nlp = spacy.load(model)

    def extract(self, text: str, embedder) -> tuple[list, list]:
        doc   = self.nlp(text)
        nodes, edges, label_to_node = [], [], {}

        # Noun chunks → ConceptNodes
        for chunk in doc.noun_chunks:
            label = chunk.text.lower().strip()
            if len(label) < 3 or label in {'it','they','this','that','we','you','i'}:
                continue
            if label in label_to_node:
                continue
            emb  = embedder.encode([label])[0].tolist()
            node = ConceptNode(
                label=label, embedding=emb,
                node_type='entity' if chunk.root.ent_type_ else 'concept',
                activation=0.7,
            )
            nodes.append(node)
            label_to_node[label] = node

        # Named entities not already captured by noun chunks
        for ent in doc.ents:
            label = ent.text.lower().strip()
            if label not in label_to_node:
                emb  = embedder.encode([label])[0].tolist()
                node = ConceptNode(label=label, embedding=emb,
                                   node_type='entity', activation=0.75)
                nodes.append(node)
                label_to_node[label] = node

        # Edges from dependency parse — one edge per adjacent chunk pair per sentence
        for sent in doc.sents:
            etype = self._infer_edge_type(sent.text.lower(), sent)
            sent_chunks = [c for c in doc.noun_chunks
                           if c.start >= sent.start and c.end <= sent.end]
            for i in range(len(sent_chunks) - 1):
                src = label_to_node.get(sent_chunks[i].text.lower().strip())
                tgt = label_to_node.get(sent_chunks[i+1].text.lower().strip())
                if src and tgt and src.node_id != tgt.node_id:
```

```
                    edges.append(ConceptEdge(
                        source_id=src.node_id,
                        target_id=tgt.node_id,
                        edge_type=etype,
                    ))
        return nodes, edges

    def _infer_edge_type(self, sent_text: str, sent) -> str:
        for m in CAUSAL_MARKS:
            if m in sent_text: return 'causal'
        for m in CONTRAST_MARKS:
            if m in sent_text: return 'contrastive'
        for m in CONSTIT_MARKS:
            if m in sent_text: return 'constitutive'
        for m in SEQUENCE_MARKS:
            if m in sent_text: return 'sequential'
        for tok in sent:
            if tok.lemma_ in CAUSAL_VERBS and tok.pos_ == 'VERB':
                return 'causal'
        return 'semantic'
```

2.4 Exit criteria

- spacy.load('en_core_web_sm') succeeds without error
- A 1-page prose document produces at least 5 ConceptNodes and 3 ConceptEdges
- All returned nodes have non-None embeddings
- Edge types are varied — not all 'semantic' — for typical prose
- Extraction of one page completes in under 15 seconds on Pi 5

3. Embedding Layer — MiniLM-L6-v2

The StubEmbedder in the existing code produces deterministic vectors from text hashes — correct for testing structural logic, wrong for semantic similarity. This section replaces it with MiniLM-L6-v2. The interface is identical: encode() takes a list of strings and returns a numpy array of unit vectors. One line changes in the entry point.

3.1 RealEmbedder implementation

```
# nbsi/embedder.py  — add this class, keep StubEmbedder for tests
from sentence_transformers import SentenceTransformer
import numpy as np

class RealEmbedder:
    """
    MiniLM-L6-v2: 80 MB · 384-dim · ~0.3 s per sentence on Pi 5.
    Semantically meaningful cosine similarity — 'cat' is close to 'dog',
    far from 'democracy'. Drop-in replacement for StubEmbedder.
    """
    DIM = 384

    def __init__(self, model_name='sentence-transformers/all-MiniLM-L6-v2'):
        self.model = SentenceTransformer(model_name)

    def encode(self, texts: list[str]) -> np.ndarray:
        # normalize_embeddings=True returns unit vectors — cosine sim = dot product
        return self.model.encode(texts, normalize_embeddings=True)

    def similarity(self, a: list[float], b: list[float]) -> float:
        va, vb = np.array(a), np.array(b)
        denom  = np.linalg.norm(va) * np.linalg.norm(vb)
        return float(np.dot(va, vb) / denom) if denom > 1e-10 else 0.0
```

3.2 Wiring into NBSISession

```
# main.py or wherever you construct the session
from nbsi.embedder import RealEmbedder          # changed
from nbsi.lifecycle.structural_library import StructuralNodeLibrary
from nbsi.session.session import NBSISession
from nbsi.config import Config

embedder = RealEmbedder()                       # was StubEmbedder()
library  = StructuralNodeLibrary()
session  = NBSISession(library, embedder, Config())
# Everything downstream is unchanged.
```

3.3 Batching rule

Always batch-encode during ingestion. A single call to encode() for 40 sentences costs roughly the same as 5 individual calls — the model startup overhead dominates per-call cost on CPU.

```
# WRONG — one call per sentence, very slow on Pi 5
for sentence in sentences:
    emb = embedder.encode([sentence])[0]

# CORRECT — single forward pass for all sentences
embeddings = embedder.encode(sentences)          # shape: (N, 384)
for sentence, emb in zip(sentences, embeddings):
    node = ConceptNode(label=sentence[:60], embedding=emb.tolist())
```

3.4 Performance on Pi 5

Operation	Time on Pi 5	Notes
Single sentence encode	~0.3 s	Acceptable for interactive query anchor finding
Batch of 20 sentences	~1.5 s	Use batching during ingestion always
Full page (~40 sentences)	~12 s	Main ingestion cost per page
Library match — 200 nodes	~50 ms	Linear cosine search. FAISS drops to ~5 ms.
Query anchor finding — 200 nodes	~50 ms	Same linear search in ConceptGraph

3.5 Exit criteria

- RealEmbedder().encode(['test sentence']) returns numpy array of shape (1, 384)
- similarity('cat', 'dog') > 0.6 and similarity('cat', 'democracy') < 0.3
- run_tests.py still shows 33/33 after swapping RealEmbedder into the session fixture
- Full page batch encode completes in under 20 seconds on target hardware

4. Synthesis Layer — llama-cpp

The synthesis layer takes the reasoning paths returned by beam search and produces natural language. The model is not reasoning from scratch — it is narrating structure that already exists. The graph found the paths. The model describes what they mean. This bounded, structured task is well within the capability of a 1–3 B parameter model.

4.1 Why a small model is sufficient here

Large models are large because they must hold world knowledge, multi-step reasoning, and language generation simultaneously. In the NBSI stack, the knowledge is in the graph and the reasoning is in the beam search. What remains for the model is: read three structured paths, understand which concepts they connect, and write a coherent paragraph. That is a language formatting task, not a knowledge task. A 1.5 B model handles it well.

4.2 Synthesiser implementation

```
# nbsi/synthesis/synthesiser.py
from llama_cpp import Llama
from nbsi.reasoning.conductivity import ReasoningPath
from nbsi.graph.concept_graph import ConceptGraph

class Synthesiser:
    def __init__(self, model_path: str, n_threads: int = 4):
        """
        model_path  : full path to the .gguf file
        n_threads   : set to the number of Pi CPU cores (4 for Pi 5)
        """
        self.llm = Llama(
            model_path=model_path,
            n_ctx=2048,          # 2048 tokens is enough for 3 paths + answer
            n_threads=n_threads,
            n_gpu_layers=0,      # CPU only — explicit
            verbose=False,
        )

    def narrate(self, query: str,
                paths: list[ReasoningPath],
                graph: ConceptGraph) -> str:
        """
        Narrate what the top paths mean in relation to the query.
        Returns grounded natural language. Does not hallucinate beyond the paths.
        """
        if not paths:
            return 'No relevant paths found in the current session graph.'

        path_text = self._format_paths(paths, graph)
        prompt    = self._build_prompt(query, path_text)

        result = self.llm(
```

```
            prompt,
            max_tokens=400,
            temperature=0.2,   # Low — stay close to the paths
            stop=['\n\n\n', '</answer>'],
            echo=False,
        )
        return result['choices'][0]['text'].strip()

    def _format_paths(self, paths, graph) -> str:
        lines = []
        for i, p in enumerate(paths[:3], 1):
            d      = p.to_dict(graph)
            chain  = ' → '.join(d['path'])
            conf   = d['conductivity']
            lines.append(f'Path {i} (confidence {conf:.2f}): {chain}')
        return '\n'.join(lines)

    def _build_prompt(self, query: str, path_text: str) -> str:
        return (
            'You are reading reasoning paths from a knowledge graph.\n'
            'Each path shows how concepts in the document connect.\n'
            'Higher confidence means stronger grounding in the source.\n\n'
            f'Question: {query}\n\n'
            f'Reasoning paths:\n{path_text}\n\n'
            'Answer based only on these paths. State what they show. '
            'Note confidence. Flag if the paths do not clearly answer the question.\n'
            'Answer:'
        )
```

4.3 Integrating synthesis with the session

```
# Wrapper function — use this in main.py
def query_and_narrate(session, synthesiser, query_text: str) -> dict:
    # Step 1: graph traversal — fast, always runs
    path_dicts = session.query(query_text)                 # < 100 ms

    if not path_dicts:
        return {'paths': [], 'answer': 'No relevant content found.'}

    # Step 2: get ReasoningPath objects for synthesiser
    query_emb = session.embedder.encode([query_text])[0].tolist()
    paths     = session._query_engine.query(query_emb)

    # Step 3: synthesis — slow, model inference
    answer = synthesiser.narrate(query_text, paths, session.graph)  # 20–60 s

    return {
        'paths':  path_dicts,   # Structured data — show user the evidence
        'answer': answer,       # Natural language — the narrated conclusion
    }
```

4.4 Expected latency on Pi 5

Model	Tokens / sec	400-token answer	RAM
Qwen2.5 0.5B Q4	~25 tok/s	~16 seconds	~800 MB
Qwen2.5 1.5B Q4	~12 tok/s	~33 seconds	~1.5 GB
Phi-3 Mini 3.8B Q4	~4 tok/s	~100 seconds	~3.5 GB

STREAMING FOR INTERACTIVE USE

For a conversational interface, stream tokens as they are generated rather than waiting for the full response. llama-cpp supports this via the stream=True parameter. The graph paths are available immediately (< 100 ms) and can be shown to the user while synthesis runs. The user sees evidence first, then narration — which is also the more honest presentation.

4.5 Exit criteria

- Llama(model_path=..., n_gpu_layers=0) loads without error
- narrate() returns a non-empty string for a hand-built 3-path input
- The response references at least one concept from the provided paths
- temperature=0.2 produces consistent answers across three runs on the same input
- Confirmed n_gpu_layers=0 — nvidia-smi or gpu-stat show zero GPU activity

5. OS Integration Pipeline

The OS integration layer watches the local file system and feeds documents into the NBSI session automatically. This is the component that makes NBSI an ambient OS assistant rather than a tool you must invoke manually.

5.1 Supported document types

Format	Extension(s)	Reader library	Notes
Plain text / markdown	.txt .md	open() — stdlib	Fast. Immediate.
PDF	.pdf	pdfminer.six	Most common document format
Word document	.docx	python-docx	Common for notes and reports
Calendar	.ics	icalendar	Events become temporal sequence nodes
Python source	.py	open() + ast	Imports and function names become entities
Email (Maildir)	no extension	mailbox — stdlib	Subject + body extracted as prose

5.2 Document reader

```
# nbsi/os_integration/reader.py
import os

def read_file(path: str) -> str | None:
    """Read any supported file to plain text. Returns None on failure."""
    ext = os.path.splitext(path)[1].lower()
    try:
        if ext in ('.txt', '.md', '.py', '.js', '.ts', '.rst'):
            with open(path, 'r', encoding='utf-8', errors='ignore') as f:
                return f.read()

        elif ext == '.pdf':
            from pdfminer.high_level import extract_text
            return extract_text(path)

        elif ext == '.docx':
            from docx import Document
            doc = Document(path)
            return '\n'.join(p.text for p in doc.paragraphs if p.text.strip())

        elif ext == '.ics':
            from icalendar import Calendar
            with open(path, 'rb') as f:
                cal = Calendar.from_ical(f.read())
```

```
            lines = []
            for component in cal.walk():
                if component.name == 'VEVENT':
                    lines.append(f"{component.get('DTSTART','')}: "
                                 f"{component.get('SUMMARY','')}. "
                                 f"{component.get('DESCRIPTION','')}")
            return '\n'.join(lines)

    except Exception as e:
        print(f'[reader] Failed to read {path}: {e}')
    return None
```

5.3 File system watcher

```
# nbsi/os_integration/watcher.py
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler
import os, threading, queue

WATCHED_EXTS = {'.txt','.md','.pdf','.docx','.py','.ics','.js','.ts'}

class NBSIFileWatcher(FileSystemEventHandler):
    def __init__(self, ingest_queue: queue.Queue):
        self.queue  = ingest_queue
        self._seen  = set()       # Debounce rapid successive events on same file

    def on_created(self, event):
        if not event.is_directory: self._enqueue(event.src_path)

    def on_modified(self, event):
        if not event.is_directory: self._enqueue(event.src_path)

    def _enqueue(self, path: str):
        if os.path.splitext(path)[1].lower() not in WATCHED_EXTS: return
        if path in self._seen: return
        self._seen.add(path)
        self.queue.put(path)
        # Allow re-ingest after 10 s if the file changes again
        threading.Timer(10.0, lambda: self._seen.discard(path)).start()


def start_watcher(watch_path: str, ingest_queue: queue.Queue):
    handler  = NBSIFileWatcher(ingest_queue)
    observer = Observer()
    observer.schedule(handler, watch_path, recursive=True)
    observer.start()
    return observer
```

5.4 Ingestion worker

```
# nbsi/os_integration/ingestion_worker.py
```

```
import queue, threading
from nbsi.ingestion.spacy_extractor import SpacyExtractor
from nbsi.os_integration.reader import read_file
from nbsi.session.session import NBSISession

class IngestionWorker:
    """
    Background thread. Pulls file paths from the queue.
    Reads the file, extracts nodes and edges, ingests into the session.
    Runs at low priority so it never blocks an active query.
    """
    def __init__(self, session: NBSISession,
                 extractor: SpacyExtractor,
                 ingest_queue: queue.Queue):
        self.session   = session
        self.extractor = extractor
        self.queue     = ingest_queue
        self._stop     = threading.Event()
        self._thread   = threading.Thread(target=self._run, daemon=True)

    def start(self): self._thread.start()
    def stop(self):  self._stop.set()

    def _run(self):
        while not self._stop.is_set():
            try:
                path = self.queue.get(timeout=1.0)
                self._ingest(path)
                self.queue.task_done()
            except queue.Empty:
                continue

    def _ingest(self, path: str):
        text = read_file(path)
        if not text or len(text.strip()) < 50:
            return
        print(f'[NBSI] Ingesting {path}')
        nodes, edges = self.extractor.extract(text, self.session.embedder)
        if nodes:
            result = self.session.ingest_graph(nodes, edges)
            print(f'[NBSI] Done — {result["nodes"]} nodes, {result["edges"]} edges')
```

5.5 Exit criteria

- read_file() returns non-empty string for a PDF, a DOCX, and a plain text file
- Creating a .txt file in the watched directory triggers ingestion within 5 seconds
- Modifying the same file a second time triggers re-ingestion after the debounce window
- Ingestion worker does not block the main thread during a long document parse

6. Orchestration — Entry Point and Persistence

6.1 Graph and library persistence

The session graph and structural library must survive reboots. Pickle is sufficient for a home deployment. For production on a Pi (where power loss can corrupt writes), replace pickle with SQLite using the stdlib sqlite3 module.

```
# nbsi/persistence.py
import pickle, os
from nbsi.graph.concept_graph import ConceptGraph
from nbsi.lifecycle.structural_library import StructuralNodeLibrary

def save(obj, path: str):
    os.makedirs(os.path.dirname(path), exist_ok=True)
    with open(path, 'wb') as f: pickle.dump(obj, f)

def load(path: str, default_factory):
    if not os.path.exists(path): return default_factory()
    with open(path, 'rb') as f:  return pickle.load(f)
```

6.2 main.py — full entry point

```
# main.py
import queue, os
from nbsi.config import Config
from nbsi.embedder import RealEmbedder
from nbsi.session.session import NBSISession
from nbsi.ingestion.spacy_extractor import SpacyExtractor
from nbsi.synthesis.synthesiser import Synthesiser, query_and_narrate
from nbsi.os_integration.watcher import start_watcher
from nbsi.os_integration.ingestion_worker import IngestionWorker
from nbsi.persistence import save, load
from nbsi.lifecycle.structural_library import StructuralNodeLibrary
from nbsi.graph.concept_graph import ConceptGraph

WATCH_PATH   = os.path.expanduser('~/Documents')
GRAPH_PATH   = os.path.expanduser('~/.nbsi/graph.pkl')
LIB_PATH     = os.path.expanduser('~/.nbsi/library.pkl')
MODEL_PATH   = os.path.expanduser('~/nbsi-models/qwen2.5-1.5b-instruct-q4_k_m.gguf')

print('[NBSI] Loading structural library...')
library = load(LIB_PATH, StructuralNodeLibrary)
print(f'       {len(library.nodes)} structural nodes')

print('[NBSI] Loading embedder...')
embedder = RealEmbedder()

session = NBSISession(library, embedder, Config())
saved_graph = load(GRAPH_PATH, ConceptGraph)
if saved_graph.node_count > 0:
```

```
    session.graph = saved_graph
    print(f'[NBSI] Restored graph: {session.graph.node_count} nodes')

print('[NBSI] Loading synthesis model (this takes ~30 s on Pi 5)...')
synth = Synthesiser(MODEL_PATH, n_threads=4)
print('[NBSI] Ready.')

ingest_queue = queue.Queue()
extractor    = SpacyExtractor()
worker       = IngestionWorker(session, extractor, ingest_queue)
worker.start()
observer     = start_watcher(WATCH_PATH, ingest_queue)
print(f'[NBSI] Watching {WATCH_PATH}')
print('[NBSI] Commands:  any question  |  spec: <statement>  |  quit')

try:
    while True:
        raw = input('\n> ').strip()
        if not raw: continue
        if raw.lower() in ('quit','exit','q'): break

        if raw.startswith('spec:'):
            r = session.speculate(raw[5:].strip())
            print(f'Speculation active: {r["spec_id"][:8]}  ({r["edges_affected"]}
edges affected)')
            continue

        print(f'[graph: {session.graph.node_count} nodes — traversing...]')
        result = query_and_narrate(session, synth, raw)

        if not result['paths']:
            print('No relevant paths found in current graph.')
            continue

        print('\nEvidence paths:')
        for p in result['paths'][:3]:
            chain = ' → '.join(p['path'])
            print(f'  [{p["conductivity"]:.2f}]  {chain}')

        print(f'\nAnswer:\n{result["answer"]}')

except KeyboardInterrupt:
    pass
finally:
    print('\n[NBSI] Saving state...')
    save(session.graph, GRAPH_PATH)
    save(library, LIB_PATH)
    observer.stop()
    worker.stop()
    print('[NBSI] Done.')
```

7. Build Order and Checkpoints

Build in this order. Each phase has an explicit checkpoint that must pass before proceeding. Do not skip ahead — each layer depends on the one below.

Phase	Task	Checkpoint	Est. time
0 — Foundation	Clone nbsi_v1_external.zip. Install networkx, numpy, scipy.	run_tests.py shows 33/33 pass	30 min
1 — Extraction	Install spaCy + model. Implement SpacyExtractor. Test on a real document.	One page → ≥ 5 nodes, ≥ 3 edges. Edge types varied. Time < 15 s.	Half day
2 — Embedding	Install sentence-transformers. Implement RealEmbedder. Swap into session.	33/33 still pass. similarity('cat','dog') > 0.6.	2 hours
3 — Synthesis	Download model. Implement Synthesiser. Test narrate() on hand-built paths.	Non-empty response that references path concepts. n_gpu_layers=0.	Half day
4 — Readers	Implement reader.py. Test read_file() on PDF, DOCX, TXT files you own.	Each format returns non-empty string.	Half day
5 — Watcher	Implement watcher.py + IngestionWorker. Point at a test directory.	Create a .txt file → ingestion triggers within 5 seconds.	Half day
6 — Persistence	Implement persistence.py. Test save/load for graph and library.	Restart process. Restored graph has same node count.	2 hours
7 — Integration	Wire main.py. Run full system. Ingest a real document. Ask a question.	Answer references concepts from the ingested document.	Half day
8 — Pi tuning	On Pi 5: set MAX_NODES=150, BEAM_WIDTH=6, n_threads=4. Profile RAM.	RAM stable under 6 GB during ingestion. Query latency acceptable.	2 hours

THE PROOF-OF-LIFE TEST

Place a document you have never queried before in ~/Documents. Wait for the terminal to show ingestion complete. Ask a question whose answer is in that document. Verify that the evidence paths reference concepts from that document and that the answer is grounded in those paths rather than hallucinated from the model's weights. If this passes, the full lightweight stack is working end to end.

8. Performance Targets and Limitations

8.1 Performance targets

Operation	Pi 5 target	Pi 4 estimate	Notes
Ingest 1 page	~60 seconds	~2 minutes	spaCy parse + batch embed. One-time cost per document.
Graph traversal (query)	< 100 ms	< 200 ms	Pure CPU arithmetic. Fast regardless of hardware.
Synthesis — 400 tokens	~35 s (1.5B model)	~70 s	Stream tokens for interactive feel.
Library match — 200 nodes	~50 ms	~100 ms	Linear search. FAISS upgrade drops to ~5 ms.
RAM at rest	~2.5 GB	~2.5 GB	OS + embedder + model + graph at 150 nodes
RAM during ingestion	~3.2 GB	~3.2 GB	Spike during batch embedding of a full page

8.2 RAM budget — Pi 5 with 8 GB

Component	RAM	Notes
OS + system processes	~800 MB	Raspberry Pi OS Bookworm baseline
Python runtime + NBSI core	~200 MB	Lightweight — no large frameworks
MiniLM-L6-v2 embedder	~200 MB	Loaded once at startup, stays resident
Qwen2.5 1.5B Q4 model	~1.5 GB	Loaded once at startup, stays resident
Session graph (150 nodes)	~50 MB	Small — graph is topology not content
Structural library (100 nodes)	~30 MB	Grows slowly across sessions
Headroom	~5.2 GB	Comfortable — room for other Pi processes

8.3 Known limitations

- spaCy edge type inference is grammatical, not semantic. It misses implied causality and metaphorical relationships. An LLM extractor produces richer edges. spaCy is the right starting point — upgrade the extractor if edge quality is insufficient for your documents.
- Synthesis latency at 35 seconds is slow for rapid conversation. It is acceptable for document analysis. Stream tokens or show paths immediately while synthesis completes.
- The structural library takes time to mature. MIN_CONFIRMATIONS requires patterns confirmed across multiple sessions on diverse documents. After roughly 20–30

sessions the library begins to meaningfully shape graph construction. A new installation starts without this advantage.

- Pickle persistence is fragile on power loss. For a Pi that may lose power, replace with SQLite (stdlib sqlite3) — wrap save() and load() in persistence.py with a transaction-safe SQLite write.
- MAX_NODES=150 is conservative for Pi 5 with 8 GB. If RAM is available and ingestion quality matters more than speed, raise it to 200 or 250. Monitor memory during ingestion with htop.

FAISS UPGRADE PATH

The StructuralNodeLibrary.match() and ConceptGraph.find_similar_nodes() methods use linear cosine search — O(N) over all nodes. This is fine up to a few hundred nodes. When the library grows beyond 500 structural nodes, replace the internals of both methods with faiss-cpu IndexFlatIP. The interface is unchanged — the rest of the system does not know or care. On Pi 5, FAISS cuts library match time from ~50 ms to ~5 ms at 500 nodes.